11-2017

APIBot: Question answering bot for API documentation

Yuan TIAN
Singapore Management University, yuan.tian.2012@smu.edu.sg

Ferdian THUNG
Singapore Management University, ferdianthung@smu.edu.sg

Abhishek SHARMA
Singapore Management University, abhisheksh.2014@smu.edu.sg

David LO
Singapore Management University, davidlo@smu.edu.sg

DOI: https://doi.org/10.1109/ASE.2017.8115628

Published in ASE 2017: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, Urbana-Champaign, IL, October 30 - November 3, pp 153-158. https://doi.org/10.1109/ASE.2017.8115628

APIBot: Question Answering Bot for API Documentation

Yuan Tian*, Ferdian Thung*, Abhishek Sharma*, and David Lo
School of Information Systems, Singapore Management University, Singapore
{yuan.tian.2012,ferdiant.2013,abhisheksh.2014,davidlo}@smu.edu.sg

***Abstract*—As the carrier of Application Programming Interfaces (APIs) knowledge, API documentation plays a crucial role in how developers learn and use an API. It is also a valuable information resource for answering API-related questions, especially when developers cannot find reliable answers to their questions online/offline. However, finding answers to API-related questions from API documentation might not be easy because one may have to manually go through multiple pages before reaching the relevant page, and then read and understand the information inside the relevant page to figure out the answers. To deal with this challenge, we develop APIBot, a bot that can answer API questions given API documentation as an input. APIBot is built on top of SiriusQA, the QA system from Sirius, a state of the art intelligent personal assistant. To make SiriusQA work well under software engineering scenario, we make several modifications over SiriusQA by injecting domain specific knowledge. We evaluate APIBot on 92 API questions, answers of which are known to be present in Java 8 documentation. Our experiment shows that APIBot can achieve a Hit@5 score of 0.706.**

***Index Terms*—API Documentation, Question Answering, Bot**

I. Introduction

When developing applications using Application Programming Interfaces (APIs), the official API documentation is one of the excellent sources for finding answers to API-related questions. Unfortunately, to find a desired piece of information, developers may need to sift through numerous pages in documentation, which is a tedious and time consuming activity. The process is even worse for APIs having a sharp learning curve [1]. As a consequence, developers often do not read API documentation [2], [3], [4]. This may cause developers to use APIs incorrectly, resulting in bugs and even security vulnerabilities [5], [6].

Finding answers in API documentation can be made much simpler through a question answering bot. The bot can simulate an expert, answering developer queries directly and reducing the need of developers to browse multiple documents to find answers. Such a bot can be particularly helpful for new or closed APIs which have few available experts. A bot can automatically learn from documentation of these new or closed APIs and help developers with their queries.

There are recent interests in creating bots for software engineering purposes. Murgia et al. created Joey, a question answering bot for StackOverflow that is able to answer questions that are asked before [7]. Storey and Zagalsky visioned the use of bots to automate tasks in software development [8]. In this paper, we build a bot that is able to answer API-related questions by analyzing API documentation. Different from Joey, our approach is able to answer questions regardless whether it has been asked before or not.

Recently, Hauswald et al. developed an open end-to-end personal assistant that they named Sirius [9]. Sirius makes use of existing state of the art technologies such as Pocketsphinx [10], Kaldi [11], and RWTH's RASR [12] for speech recognition, OpenEphyra [13] for question answering, and SURF [14] for image detection. SiriusQA, i.e., the QA system inside Sirius, works by relying on a set of question and answer patterns. Question patterns are used to extract important phrases from an input question, while answer patterns are used to rank candidate answers extracted from a text corpus. SiriusQA has been shown to work well with general knowledge questions. However, it is unknown whether SiriusQA would still work for domain-specific questions like API-related questions due to its use of *general* question and answer patterns. These patterns do not capture API domain knowledge.

Faced with SiriusQA's inherent limitations for the API domain, we are motivated to build APIBot, a specific question answering bot for API documentation. We address limitations of SiriusQA to make it work well for API documentation. First, we naturalize API documentation to make hidden structural information appear in the form of natural language sentences. Second, we create API-specific question patterns to make SiriusQA understand important parts of API questions and categorize them. Third, we modify SiriusQA to consider API-specific answer patterns and terms. Finally, we learn answer patterns by building a probabilistic model for each answer category. These answer patterns require a smaller training corpus than SiriusQA, which needs to learn answer patterns in form of regular expressions from a large set of question-answer pairs.

To evaluate the performance of APIBot, we collect a benchmark by first inviting a group of people to ask questions about JDK 8 API documentation. We then invite another group of people to provide the ground truth answers for the collected questions. Our experimental result shows that APIBot can achieve a Hit@5 score of 0.706 on the evaluation dataset.

The rest of this paper is structured as follows. Section II describes background knowledge of SiriusQA and its limitations when applied on API documentation. Section III introduces the design of APIBot. Section IV presents our evaluation

*The first 3 authors have contributed equally to the work.

methodology and results. Section V describes related studies. Section VI gathers the conclusions derived from this work and presents future work.

II. SiriusQA and Its Limitations

A. Overview of SiriusQA

Sirius is the leading state-of-the-art open source intelligent personal assistant (IPA) system. It consists of three major components: automatic speech recognition (ASR), question-answering (QA) and image matching (IMM). In this work, we enhance the question-answering component of Sirius, henceforth referred to as SiriusQA. SiriusQA is based on the open source OpenEphyra system [13]. The system incorporates a number of NLP algorithms to effectively answer natural language questions.

SiriusQA consists of three major components: question interpretation, document search, and answer selection. We describe them briefly below:

Question Interpretation. In *question interpretation* component, SiriusQA identifies the *target* (i.e., the entity that a question asked about), *contexts* (i.e., the qualifying attributes of the target), and *category* of a question based on a pre-defined set of manually constructed question patterns. A *question pattern* is a regular expression with placeholders to identify target and context phrases from questions. These patterns are grouped into different categories.

For example, consider the following question: "*What medal has Joseph Schooling won in 2016 Olympics?*". Given the following question pattern , "*What medal has* $\langle target \rangle$ *won in* $\langle context \rangle$*?*",we can find that the *target* is "*Joseph Schooling*" and the *context* is "*2016 Olympics*".

Document Search. In *document search* component, relevant documents matching a query are returned. The query is generated from the *target*, *context*, and *keywords* extracted by the *question interpretation* component.

Answer Selection. In *answer selection* component, candidate answers are identified from returned documents by matching series of consecutive words in the documents with answer patterns. The answer patterns are again in the form of regular expressions, with placeholders to identify specific text that is to be returned as an answer. A confidence score is assigned to each answer pattern, and this score is automatically inferred based on a training set of question-answer pairs.

B. Limitations of SiriusQA

SiriusQA has been demonstrated to work well in answering general questions by analyzing a corpus of textual articles. However, it is unlikely to work well in answering API-specific questions by analyzing API documentations, due to the following limitations.

Limitation 1: SiriusQA only handles unstructured textual documents. API documentation, on the other hand, is a structured document. For example, in an API documentation of a library written in an object oriented language (e.g., a Javadoc page), it would have pages explaining about classes. Throughout the page, there would also be links to other pages in the API documentation representing inheritance or other kinds of relationships. Unfortunately, such structural information is fully ignored in SiriusQA.

Limitation 2: SiriusQA uses a set of manually crafted question patterns, which are designed for general knowledge questions rather than API-specific questions. An example of general knowledge question that SiriusQA can handle is *"What is the name of the actor who star in Titanic?"*. This question is far from questions developers might ask about an API. In addition, each question pattern is assigned to a category in SiriusQA, such as `author`, `capital`, etc. However, none of the categories are related to the type of knowledge that are commonly contained in API documentation.

Limitation 3: Answer pattern learning in SiriusQA requires a large amount of manually created training data. However, manually creating a large amount of API-specific question-answer pairs is difficult and time consuming. StackOverflow records many questions (including those that are related to APIs) and their corresponding answers. However, converting StackOverflow data into something that SiriusQA can handle requires expensive manual cleaning step. Thus, a strategy is needed to learn answer patterns from small amount of data.

Limitation 4: SiriusQA does not have any knowledge about software terms, e.g., it does not know that *ArrayList* is a class in Java 8. It thus cannot differentiate software-specific words from other words, and cannot use domain-specific heuristics to return better answers.

III. Plugging In Software Knowledge to SiriusQA

A. Overview

We have built our system APIBot on top of SiriusQA by modifying and enhancing some of its components. The overall framework of APIBot is shown in Figure 1, with boxes shaded in grey representing our new or enhanced components. It consists of two major parts: *Domain Adaptation*, and *Enhanced SiriusQA*. A brief overview of each part is given below.

Domain Adaptation. The Domain Adaptation component enables APIBot to understand software engineering questions and produce appropriate answers through a one-time training process. At the end of the process, this component produces 3 outputs, i.e., *Enhanced API Documentation*, *Question Patterns* and *Answer Patterns*.

The Enhanced API Documentation consists of natural language sentences which describe all the structured and unstructured information present in the original API documentation. An Enhanced API Documentation enables us to take into consideration the structure of an API documentation and hence addresses *limitation 1*.

Question Patterns, which consist of regular expressions with placeholders to identify target and context words from questions, are another output. These expressions are grouped into different categories capturing different kinds of knowledge stored in an API documentation [15]. Our question patterns, which are grouped into these domain-specific categories, address *limitation 2*.

The Domain Adaptation component also outputs a probabilistic model for each category of questions. Each model is able to assign a probability to a candidate answer sentence in the Enhanced API Documentation based on its likelihood to be an answer to a target question of a given category. These models constitute the third output, i.e., Answer Patterns. Our answer patterns are different from those of SiriusQA; rather than being regular expressions, they are probabilistic models. Both SiriusQA and APIBot automatically learn answer patterns from a set of question-answer pairs. We choose to learn a probabilistic model over regular expressions since the latter tends to overfit, especially when we only have a small set of training data. Our probabilistic answer patterns thus addresses *limitation 3*.

Enhanced SiriusQA. All of the three outputs produced are then plugged in to an enhanced SiriusQA. We enhance SiriusQA by modifying its Answer Selection component to be able to use the *probabilistic* Answer Patterns generated by the Domain Adaptation component and address *limitation 4*. The enhanced SiriusQA receives a question and produces an answer by executing the following process:

- The Question Interpretation component interprets the question by matching it against each of the Question Patterns generated by the Domain Adaptation component. It produces a query for the Document Search component and a question category. It follows the same process described in Section II-A.
- The Document Search component takes in a query and outputs a ranked list of documents that are then fed to the Answer Selection component. It follows the same process described in Section II-A.
- The Answer Selection component first breaks returned documents into candidate answer sentences. It then ranks candidate answer sentences using the answer pattern of the identified question category and a customized keyword matching strategy that addresses *limitation 4*.

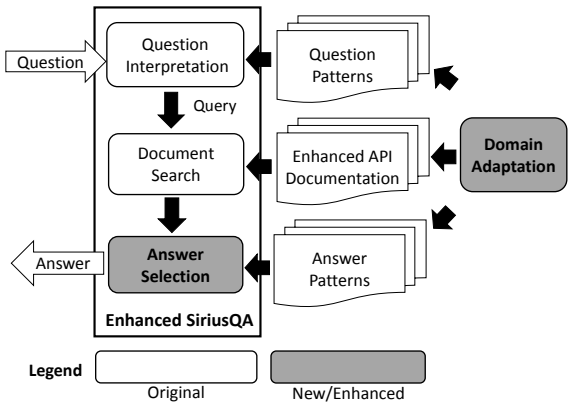

Fig. 1. APIBot Framework

B. Domain Adaptation

The Domain Adaptation is completed through two steps. Each step is described in detail below.

1) Step 1: Document Naturalization: In an API documentation, much information is stored in the document structure. For example, the structure indicating API elements (e.g., classes, methods, constructors, fields, etc.) is described by different description block. Unfortunately, as discussed in Section II, SiriusQA cannot deal with structured documents. Simply flattening structured documents into regular text files would not work since essential pieces of information stored in the document structure would be lost. To tackle this challenge, we convert the structured information in API documentation to natural language sentences which can be understood by SiriusQA. We design a number of rule-based heuristics that make use of the regularities in the structure of API documentation to create natural language sentences that explicitly describe the implicit information stored in the API documentation structure. In this work, we focus on Javadoc API documentation which is one of the most popular API documentation formats. Similar rule-based heuristics can be designed for other API documentation formats, which we leave as future work.

The Documentation Naturalization step takes in a Javadoc API documentation and converts it to a collection of natural language documents which we refer to as enhanced API documentation. This step iterates over all documentation pages (for classes and interfaces) and then transforms implicit information stored in the Javadoc structure into plain sentences. For instance, information from the *inheritance* block in a Javadoc HTML document will be transformed into a sentence like "*ArrayList extends AbstractList.*".

2) Step 2: Question-Answer Pattern Generation: In this step, we generate question and answer patterns from a set of training question-answer pairs. Figure 2 shows a overview of this step. The question patterns are generated through an annotation process, while the answer patterns are generated through an automated inference process. Both patterns are grouped according to a predefined category.

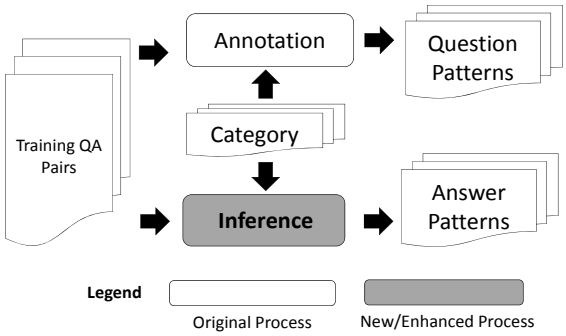

Fig. 2. Question-Answer Pattern Generation

Question Pattern Annotation: We consider a similar methodology as SiriusQA to create question patterns from a set of training question-answer pairs (See Section II-A). The major difference is that we consider a new set of domain specific categories. SiriusQA has categories that are not suitable for API documentation. We use the categories related to API domain by following the work of Maalej and Robillard [15]. Excluding *non-information*, they defined 11 types of domain

TABLE I
TAXONOMY OF QUESTION CATEGORIES

Category	Description	Example Question
Code Example	Questions requesting a code example for a specific task or functionality.	*Do you have a sample code to append a string to a StringBuilder?*
Concepts	Questions related to explanation of a particular API entity or domain concepts.	*What is StringBuilder?*
Control Flow	Questions related to actions which need to follow a particular execution order.	*What happens when you call notifyObservers() method in Observable class?*
Directives	Questions asking about allowed or disallowed contracts provided by API elements.	*Can I control the number of threads to execute CompletableFutures?*
Environment	Questions about API version, licences, compatibility issues etc.	*What are the new things provided by arraylist class in this version of API?*
Functionality and Behavior	Question asking about a feature or functionality provided by the API.	*What will happen if there are multiple duplicate objects exist when using remove(Object o) of arraylist?*
Patterns	Questions describing a specific operation and query about how the API can be used to complete the operation.	*How can I add element into ArrayList?*
Purpose and Rationale	Questions related to a design and/or purpose rationale of an API element.	*Why is HashMap not thread safe?*
Quality Attribute and Intenal Aspects	Questions related to non-fucntional aspects of an API.	*What are the two parameters of an instance of HashMap that affect its performance?*
Reference	Questions asking for a reference or extra information related to an API element.	*What classed or interfaces I can read about which are most similar to ArrayList?*
Structure and Relationships	Questions pertaining to the hierarchy or organization of API.	*What are the implemented interfaces of HashMap?*

knowledge inside API documentation. We create a category for each one of the knowledge types. Table I shows the definition and a sample question for each category. Given a new question, APIBot makes use of annotated question patterns to decide the category, target and contexts of a new question.

Answer Pattern Inference: This step takes as input a training set of question-answer pairs from a given category. For every question, its *target* phrase has been identified, and this *target* must appear in the corresponding answer.

We abstract an answer into a generalized answer by following these steps: (1) We replace the *target* phrase with $\langle target \rangle$; (2) We identify API elements, e.g., class names, package names, etc., from each answer sentence and replace them with their types, e.g., $\langle class \rangle$, $\langle interface \rangle$, $\langle package \rangle$; (3) We replace every be-verb (e.g., is, are, etc.) with $\langle BE \rangle$ and every verb with $\langle VERB \rangle$[1]; (4) We use the symbol "#" to represent the starting point of a sentence; (5) All other words are kept unchanged.

Next, we reduce the size of each generalized answers by considering N words before and after the $\langle target \rangle$. For instance, in the experiment, N is set to 3 by default, then every generalized answer is reduced to a sequence of 7 words, including 3 words before and 3 words after the $\langle target \rangle$. We refer to this sequence of words as a reduced answer.

For each question category, we then create an answer pattern based on reduced answers of the category. An answer pattern is a probability model consisting of a collection of word probability distributions. Each distribution corresponds to one of the N positions before and after $\langle target \rangle$. It is created by computing the relative frequency of words appearing on the corresponding position in the reduced answers.

Given a question which has been interpreted by the Question Interpretation component, question category and $\langle target \rangle$ are identified. Answer pattern of the corresponding question category is then used to assign a probability score to a candidate answer sentence. Each candidate answer sentence is first abstracted to a generalized answer and then reduced to a sequence of $2N + 1$ words, with the $\langle target \rangle$ in the center. The candidate answer sentence's score is calculated by multiplying probabilities of words appearing before and after the $\langle target \rangle$. These probabilities are inferred from the learned answer pattern of the corresponding category.

[1]We identify verbs by using a part-of-speech (POS) tagger

C. Answer Selection

The output of *Document Search* component is a ranked list of documents that contain query keywords. We consider only top 100 documents for finding the correct answer to a given question. From these documents, we extract candidate answer sentences and rank them. The answer ranking process contains three steps: 1) document splitting and sentence filtering, 2) sentence scoring, and 3) sentence ranking. We describe the details of these steps below.

1) Step 1: Document Splitting and Sentence Filtering: We split the retrieved documents into sentences. We consider that a new sentence begins when a dot sign is followed by whitespace(s) and the following word starts with a capitalized first character. Next, we remove sentences that are unlikely to be an answer to the question. In particular, we check if names of classes or interfaces described in the API documentation appear in the question. If the names of such classes or interfaces appear, then we remove sentences that do not appear in the Javadoc API pages for those classes or interfaces, except those that contain the names.

2) Step 2: Sentence Scoring: After retrieving documents, splitting them into sentences, and filtering out-of-scope sentences, we now have candidate answer sentences. To rank these sentences, we use two scoring strategies: *Pattern-Specific* and *Domain-Specific*. If the question category is known, we compute two scores for each candidate answer sentence using the two strategies. These scores are combined together into a

final score. However, if the category is not known, we only run the domain-specific strategy. Both scoring strategies and the score combination strategy are described below.

Pattern-Specific Scoring. We use the answer pattern corresponding to the category of an input question to assign a score to a candidate sentence. The answer pattern is a probabilistic model described in Section III-B2.

Domain-Specific Scoring. While an answer pattern captures the general pattern of nearby words surrounding a question's *target*, it ignores words that are located some distance away from the *target*. These words may include important domain-specific words. In this work, we consider all class and interface names in a target Javadoc API documentation as domain-specific words. All other words are considered general words.

Next, we match the domain-specific and general words in the question with the words in the answer candidate. A matching with a domain specific word is given a higher weight and boosts the score higher compared to that of a general word. Based on the matching, we compute the following domain-specific score $DMScore$:

$$DMScore = \frac{w_1 \times \#GenWord + w_2 \times \#DomainWord}{\#Words}$$

where $GenWord$ and $DomainWord$ are the number of matched general and domain words, respectively. $\#Words$ is the total number of words. w_1 and w_2 are the weights for general and domain words, respectively. We set w_1 and w_2 to be 1 and 2, respectively. Consequently, we boost the score by giving a matching of a domain-specific word twice the importance of that of a general word.

3) Step 3: Score Normalization and Ranking: If a question category can be inferred, for each answer candidate, we have two scores, and these scores need to be unified. To unify the scores, we first normalize them. Next, we combine the normalized scores as follows:

$$FinalScore = \alpha \times PMNormScore + \beta \times DBNormScore$$

In the above equation, $PMNormScore$ and $DBNormScore$ are the normalized pattern-specific and domain-specific scores, respectively. By default, we consider both scores to be equally important so we set both α and β to be 0.5.

IV. Preliminary Experiments and Results

A. Dataset

We select Javadoc of Java Development Kit 8 as our API documentation corpus as it is one of the largest and most popular APIs. As most Java API documentation follows the same structure we believe results achieved on current dataset should be generalizable enough for other Java APIs as well. For training data, we create our own question patterns for categories in Table I. We create the questions by figuring out varying ways one can ask a question about a particular category. We also make sure that our questions have answers inside the API documentation.

For evaluation data, we ask 4 PhD students and 2 Java developers, all with more than 5 years of programming experience in Java to ask questions about each category with a requirement that the answer to each question should be found in an API documentation page. We collect a total of 100 questions: 10 for *Concept* category and 9 for each of the remaining categories. After the collection of questions, we assemble a team of 2 PhD students. We give the collected 100 questions to this team, who are required to go through all the questions, and then find a sentence in the API documentation (both enhanced and original) which could be considered as the correct answer for each question assigned. Both the team members had to discuss and come to an agreement before coming up with sentence(s) that constitute as an answer to each question. For 8 of the questions, the team cannot properly understand them or find sentences in the documentation that answer them. So, in the end, we had a total of 92 question answer pairs, and we use this data as the ground truth to evaluate the performance of APIBot. Our dataset is of similar size as used in evaluating other specialized QA systems, e.g., [16]. Although it is not very large, it covers all the categories identified by Maalej and Robillard [15], which we believe to be sufficient for preliminary experiments.

B. Experiment Setting

Evaluation Metric: In the experiment, we query each question and record the returned answers from APIBot. By default, APIBot returns five possible answers per question. We evaluate the answers by matching them against the ground truth. To measure the effectiveness of APIBot in answering questions correctly, we use $Hit@N$ metric [17], [18], [19], which measures the percentage of questions for which we are able to find the correct answer in the top N (e.g., 5) returned answers.

Baselines: We consider 3 baselines to compare with APIBot.

Baseline 1: It is the original SiriusQA system [9]. As the original SiriusQA system returns only one top ranked answer for each question, we modify it to return the top 5 ranked answers so that it can be compared against APIBot using the $Hit@N$ evaluation criteria discussed above (for N=5). For this baseline system, we use the unnormalized text extracted from original API HTML documents as the knowledge source.

Baseline 2: This baseline consists of the original SiriusQA system using the Enhanced API documentation as the knowledge source. We use this baseline to measure the performance benefits provided by the *Question-Answer Pattern Generation* step and enhanced Answer Selection component.

Baseline 3: This baseline is our APIBot system which uses the original API documentation as the knowledge source (rather than the Enhanced API Documentation). This baseline is used to measure the performance benefits provided by the *Document Naturalization* step discussed in Section III.

C. Research Questions

RQ1: How effective is our approach in answering the questions related to API documentation?

Table II shows the $Hit@5$ score of APIBot system as compared to *Baseline 1*. The $Hit@5$ score of APIBot is 70.6% as compared to 2.2% for the baseline system. Thus, the result shows that APIBot greatly improves the performance of *Baseline 1* (SiriusQA) showing the value of injecting domain knowledge to it. Without domain knowledge, Sirius cannot make use of question patterns and unable to find information that is encoded in documentation structure.

TABLE II
PERFORMANCE OF APIBOT AND SIRIUSQA

Approach	Hit@5
APIBot	70.6%
Baseline 1 (Original SiriusQA)	2.2%

RQ2: How important are Document Naturalization and Answer Pattern Inference steps for accurately answering questions related to API documentation?

The results of of our experiment w.r.t. RQ2 are shown in Table III. It shows that *Baseline 2*, which performs *Document Naturalization* step before using SiriusQA, does not result in performance gain. This is because SiriusQA cannot identity *targets* and *contexts* for API related questions. On the other hand, removing *Document Naturalization* step from APIBot results in a big performance drop (i.e., from 70.6% to 18.5%). Thus, the two steps need to be performed together to build an effective question answering system for API documentation.

TABLE III
BENEFIT OF MAIN STEPS OF APIBOT

Approach	Hit@5
Baseline 1 (Original SiriusQA)	2.2%
Baseline 2 (SiriusQA + Enhanced API Doc.)	2.2%
Baseline 3 (EnhancedSiriusQA + Original API Doc.)	18.5%
APIBot (EnhancedSiriusQA + Enhanced API Doc.)	70.6%

V. RELATED WORK

Murgia et al. developed an experimental bot to answer questions on StackOverflow [7]. It was trained to handle simple (and reoccurring) questions related to git error messages, and provide answers based on previous solutions to similar problems. The main focus was to understand how the bot is perceived when presented as a person versus as a bot. Storey and Zagalsky suggested that we could use bots to automate many tasks in software development [8]. There also has been work on retrieving relevant information from API documentation [1], [20]. Our work enriches the above mentioned studies by proposing a support bot to help in answering questions related to API documentation.

In information retrieval and natural language processing domain, many work has been done in building general-purpose QA systems, e.g., [13], [21]. In this paper, we propose a domain-specific QA system by incorporating domain knowledge to customize a general-purpose QA system.

VI. CONCLUSION AND FUTURE WORK

In this paper, to help developers find answers to API-related questions, we developed the first question answering bot for API documentation namely APIBot. APIBot is built on top of SiriusQA. We made various efforts in adapting SiriusQA for API documentation. Our preliminary experiment with 92 API-related questions demonstrates that APIBot can achieve a promising Hit@5 score of 0.706, and highlights the value of our adaptation strategies.

As a future work, we plan to pursue the following directions. First, we would like to build a larger benchmark data from various API documentations. Second, we would like to investigate possibility to reuse answer patterns learned from one API documentation for another. Third, we want to investigate the possibility of using other underlying general-purpose QA systems beyond SiriusQA by following similar adaptation strategy presented in this paper. Fourth, we also plan to develop additional strategies to better handle structural information inherent in API documentation. The strategy that we employ in the document naturalization step may not be the optimal one.

REFERENCES

[1] J. Stylos and B. A. Myers, "Mica: A web-search tool for finding api components and examples," in *VL/HCC*, 2006.
[2] H. Zhong, L. Zhang, T. Xie, and H. Mei, "Inferring resource specifications from natural language API documentation," in *ASE*, 2009.
[3] R. Pandita, X. Xiao, H. Zhong, T. Xie, S. Oney, and A. Paradkar, "Inferring method specifications from natural language API descriptions," in *ASE*, 2012.
[4] D. G. Novick and K. Ward, "Why don't people read the manual?" in *SIGDOC*, 2006.
[5] M. Kajko-Mattsson, "A survey of documentation practice within corrective maintenance," *EMSE*, 2005.
[6] S. Ma, D. Lo, T. Li, and R. H. Deng, "Cdrep: Automatic repair of cryptographic misuses in android applications," in *AsiaCCS*, 2016.
[7] A. Murgia, D. Janssens, S. Demeyer, and B. Vasilescu, "Among the machines: Human-bot interaction on social Q&A websites," in *CHI*, 2016.
[8] M.-A. Storey and A. Zagalsky, "Disrupting developer productivity one bot at a time," in *FSE*, 2016.
[9] J. Hauswald, M. A. Laurenzano, Y. Zhang, C. Li, A. Rovinski, A. Khurana, R. G. Dreslinski, T. Mudge, V. Petrucci, L. Tang *et al.*, "Sirius: An open end-to-end voice and vision personal assistant and its implications for future warehouse scale computers," in *ASPLOS*, 2015.
[10] D. Huggins-Daines, M. Kumar, A. Chan, A. W. Black, M. Ravishankar, and A. I. Rudnicky, "Pocketsphinx: A free, real-time continuous speech recognition system for hand-held devices," in *ICASSP*, 2006.
[11] D. Povey, A. Ghoshal, G. Boulianne, L. Burget, O. Glembek, N. Goel, M. Hannemann, P. Motlicek, Y. Qian, P. Schwarz *et al.*, "The Kaldi speech recognition toolkit," in *ASRU*, 2011.
[12] D. Rybach, S. Hahn, P. Lehnen, D. Nolden, M. Sundermeyer, Z. Tüske, S. Wiesler, R. Schlüter, and H. Ney, "RASR-The RWTH Aachen university open source speech recognition toolkit," in *ASRU*, 2011.
[13] N. Schlaefer, J. Ko, J. Betteridge, M. A. Pathak, E. Nyberg, and G. Sautter, "Semantic extensions of the Ephyra QA system for TREC 2007." in *TREC*, 2007.
[14] H. Bay, T. Tuytelaars, and L. Van Gool, "Surf: Speeded up robust features," in *ECCV*, 2006.
[15] W. Maalej and M. P. Robillard, "Patterns of knowledge in api reference documentation," *TSE*, 2013.
[16] O. Tsur, M. de Rijke, and K. Sima'an, "Biographer: Biography questions as a restricted domain question answering task," *in ACL*, 2004.
[17] R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry, "Improving bug localization using structured information retrieval," in *ASE*, 2013.
[18] P. Runeson, M. Alexandersson, and O. Nyholm, "Detection of duplicate defect reports using natural language processing," in *ICSE*, 2007.
[19] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun, "An approach to detecting duplicate bug reports using natural language and execution information," in *ICSE*, 2008.
[20] C. Treude, M. P. Robillard, and B. Dagenais, "Extracting development tasks to navigate software documentation," *TSE*, 2015.
[21] X. Yao, B. Van Durme, and P. Clark, "Automatic coupling of answer extraction and information retrieval." in *ACL*, 2013.